Robotics – Exercise 3 – Report

Shlomi Ben-Shushan & Yiftach Neuman

Summery

In this report we will explain our implementation preference of the algorithm that let the Krembot navigate from the starting position $(1,1)$ to the goal position $(-2,-2)$. We will describe the setup and loop functions, the positions sampling method, the distance metric, the use of KNN and KD-Tree data structure, the shortest path algorithm, and the robot's feedback-control. Afterwards we will report the success of 5 random seeds $(1,3,6,8,16)$ we used to test our navigation algorithm.

Algorithm Description

The algorithm is divided to the setup algorithm (planner) and the loop function (execution).

Setup: The setup algorithm follows the following steps:

1. Let s be the starting position $(1,1)$ and let t be the goal position $(-2,-2)$.
2. Inflate each obstacle in the grid w.r.t the robot size (we set it as constant 0.2).
3. Lower the resolution of the grid such that a cell in the low-resolution grid represents multiple cells in the high-resolution grid, and there is 1 in each low-resolution cell that represents a group of high-resolution cells that one of them contains 1 (and 0 $o.w$).
4. Sample 12,000 positions from the arena. The number of samples should be changed w.r.t the size of the arena.
5. Insert each sampled position to a KD-Node.
6. Create KD-Tree out of the KD-Nodes.
7. Calculate adjacent-matrix using KNN which use the KD-Tree for better performance.
8. Find the shortest path between s and t using Dijkstra and the adjacent-matrix.

Loop: the algorithm gets the shortest path from s to t and follows the following steps:

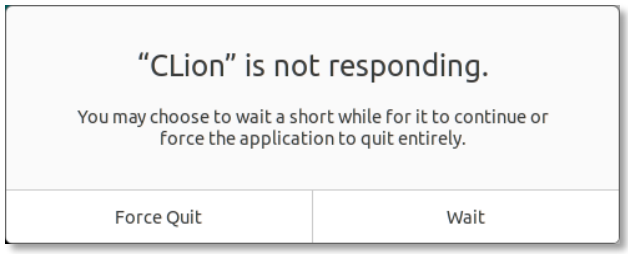
1. If current position is goal – then stop (goal reached).
2. Find the next position.
3. Turn until positioned toward the next position.
4. Drive to the next position.
5. Back to 1.

Sampling Method: we used uniform sampling of positions in order to be able to find the shortest path in different arenas without knowing the positions of the obstacles in advance. We tried to sample different numbers of positions and decided that sampling size of $12{,}000$ position is a good practice for an arena at size 250×250.

Examples:

- Sample 2 position: the algorithm couldn't find a path from s to t due to many connected components.
- Sample 6,000 positions: the algorithm succeeds to find path from s to t, but the robot lingered between adjacent obstacles so we assume that in an arena with a lot of obstacles the robot's performance might be poor.

- Sample 12,000 positions: the algorithm succeeds to find path from s to t in all of our tests. Note that it can take a while for the algorithm to calculate the path, and sometimes the program causes the following error message:



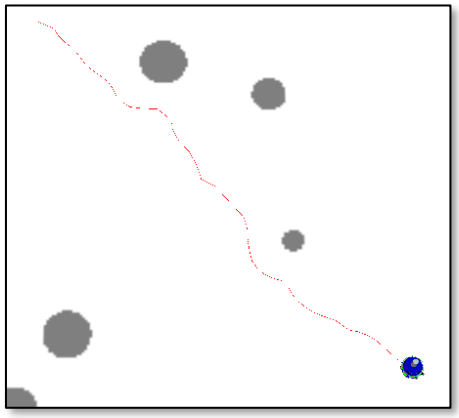
  It is okay, let the algorithm run and click "Wait" if the error message pops. You will see the robot navigate successfully as soon as the planning algorithm will finish.

Distance Metric: we have implemented two kinds of metrics:

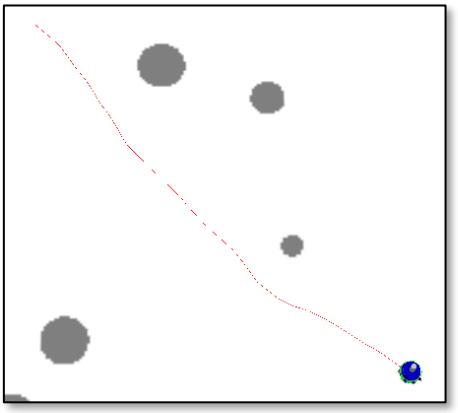
- L1 Metric – Manhattan Distance:
  Calculate the distance between points (x_1, y_1) and (x_2, y_2) via $|x_1 - x_2| + |y_1 - y_2|$.
- L2 Metric – Euclidean Distance:
  Calculate the distance between points (x_1, y_1) and (x_2, y_2) via $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

Comparison of results:

L1 Metric Result



L2 Metric Result



It seems that better performance is achieved by using the L2 metric, i.e., the robot navigates from $(1,1)$ to $(-2,-2)$ in a shorter path. This makes sense, since the L1 metric calculates the Manhattan Distance while the robot can move at any angle it chooses (i.e., also diagonally, not just up, down, right, and left). Hence, we preferred using L2 distance metric.

Data Structure: In order to calculate the adjacent-matrix, we needed to run KNN algorithm and to do so in good performance, we needed a unique data structure that stores information about points in the space (on the grid) in such a way that would fit the KNN algorithm and allow it to run quickly. The data structure that allows this is KD-Tree.

From Wikipedia:

> *In computer science, a k-d tree (short for k-dimensional tree) is a space-partitioning data structure for organizing points in a k-dimensional space. k-d trees are a useful data structure for several applications, such as searches involving a multidimensional search key (e.g., range searches and nearest neighbor searches) and creating point clouds. k-d trees are a special case of binary space partitioning trees.*

For further reading: https://en.wikipedia.org/wiki/K-d_tree

We used an implementation of KD-Tree from the internet: https://github.com/cdalitz/kdtree-cpp

KNN Calibration: We were looking for a good hyper-parameter for KNN, i.e., a number k that let the KNN associate enough neighbors for each position, so that running Dijkstra on the resulting graph will lead to a path from the starting position to the goal. On the one hand, if we use too small k, the algorithm may advance in very small steps or even won't find any path to the goal. On the other hand, if we use too large k, the algorithm may take a very long time to run. So, we have tried different ks and decided that $k = 30$ is a fair choice.

For example:

- In $k = 2$, the algorithm didn't find a path to the goal.
- In $k = 10$, the algorithm found a path, but the robot advanced in very small steps.
- In $k = 30$, the algorithm found a path, and the robot advanced in reasonable steps.

Local Planner: Because of the robot is given a task once (navigate from $(1,1)$ to $(-2,-2)$), its local planner is implemented in the setup function as described above.

The planner works as follows: $define\ s,t \Longrightarrow inflate\ obstacles \Longrightarrow lower\ resolution \Longrightarrow sample\ positions \Longrightarrow create\ KDNode\ for\ each\ position \Longrightarrow create\ KDTree \Longrightarrow Run\ KNN\ to\ get\ AdjMatrix \Longrightarrow Run\ Dijkstra\ to\ find\ shortest\ path\ from\ s\ to\ t.$

To make the robot able to receive new tasks and plan their execution in running time, it will be necessary to separate the part of the local planner from the setup function into a separate code section (suppose a function) and call that section each time a new task is received. In reality, it is better to run this code in a separate thread to avoid the robot from being stop whenever it is planning a new task.

Shortest-Path Algorithm: In the query phase, we decided to use Dijkstra algorithm for finding the shortest path from a sampled position very close to the starting position to a sampled position very close to the goal.

Feedback-Control: The planner gives the robot a path consists of waypoints (positions). In each waypoint, the robot calculates the angle to the next waypoint and start turning in angular speed of 30 or -30 (depending on which side the rotation time will be shorter), until it is positioned toward the next waypoint with error angle of $\pm 5°$. Then, the robot start driving until it reaches a position "close enough" to the next waypoint.

Speed-Angle Trade-off: there is a trade-off between the angular speed when the robot is turning, to the error angle. We could deny any error angle, and makes the robot drive only when it is positioned straight towards the next waypoint, but in this way the angular speed should be very slow because otherwise the robot may miss the perfect angle and will have to complete another circle (and then again it is possible to miss the perfect angle). As we do not want a very slow angular speed and neither a wide error angle, we had to find numbers that will work well together. In the end we determined those numbers (hard-coded) as $angularSpeed = 30$ and $errorAngle = 5°$.

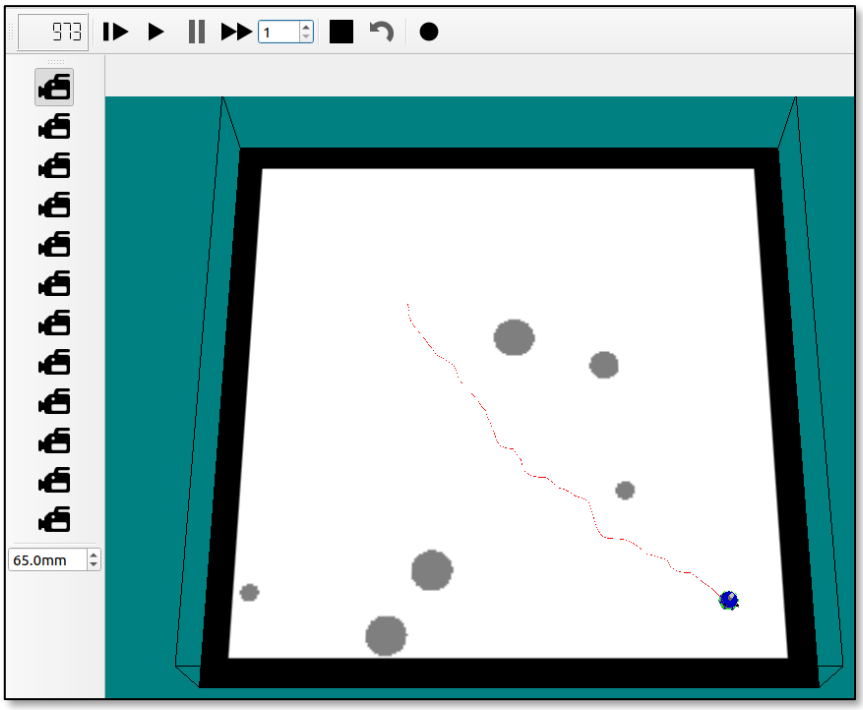
Reaching Next Waypoint: Now, we still have to explain what is it means "close enough to the next waypoint". When the robot is driving to the next waypoint that represented as a point in space, it is not reasonable that the robot will reach the exact position perfectly due to small angle errors and stopping times. Hence, we had to find a number that means "close enough" to the waypoint, and we determined this number as 2. This is reflected in the code through the function *close_enough()* that calculates the distance (using L2 metric) and return *((0 <= distance) && (distance <= 2))* as Boolean.

Success Report

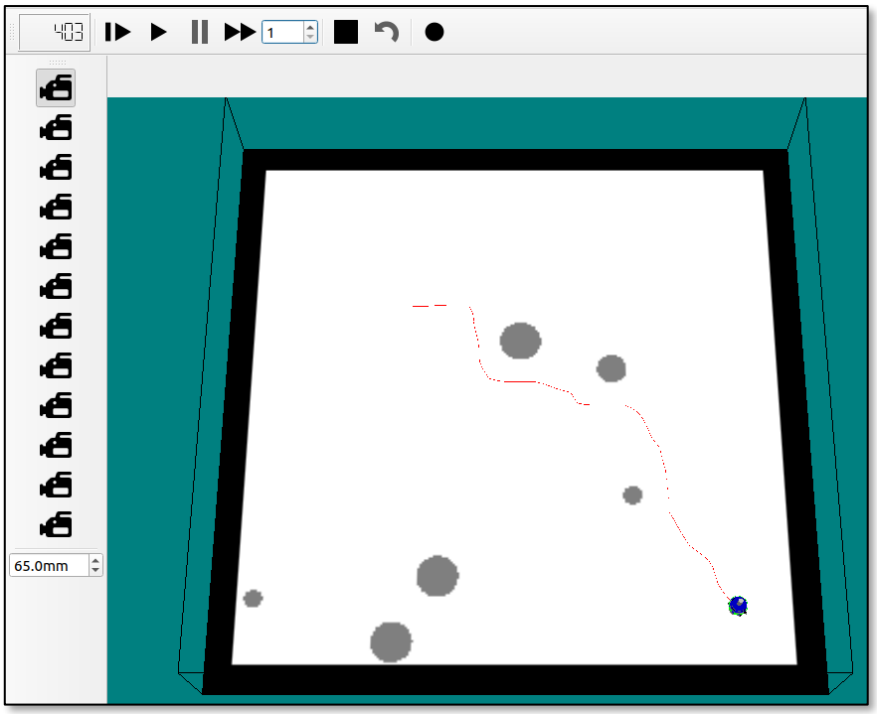
Seed Report 1:

In this experiment we set $random\ seed = \mathbf{1}$.

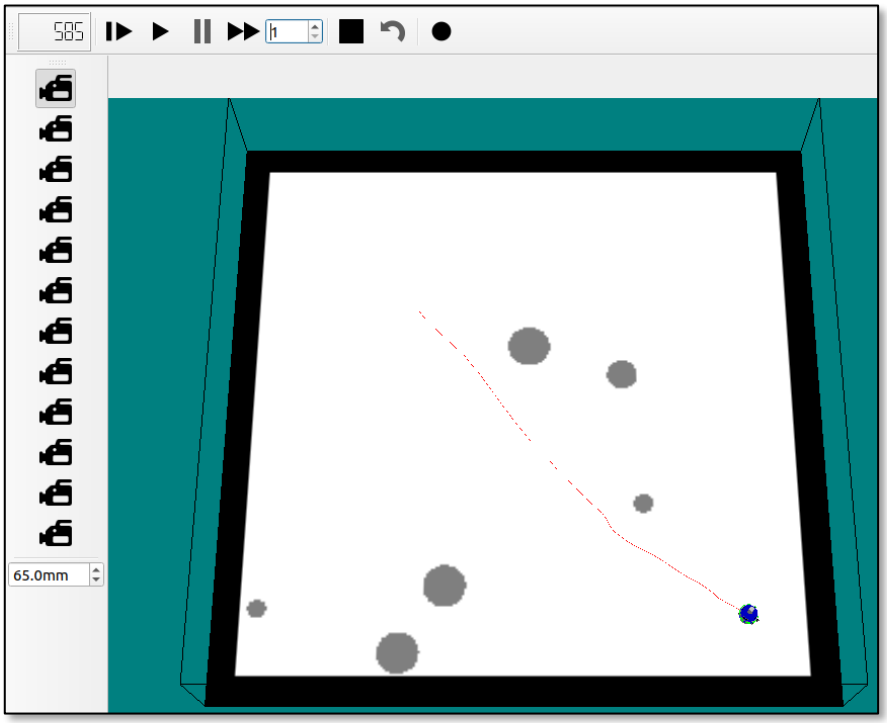
$Distance\ Metric = L1, k = 10$:



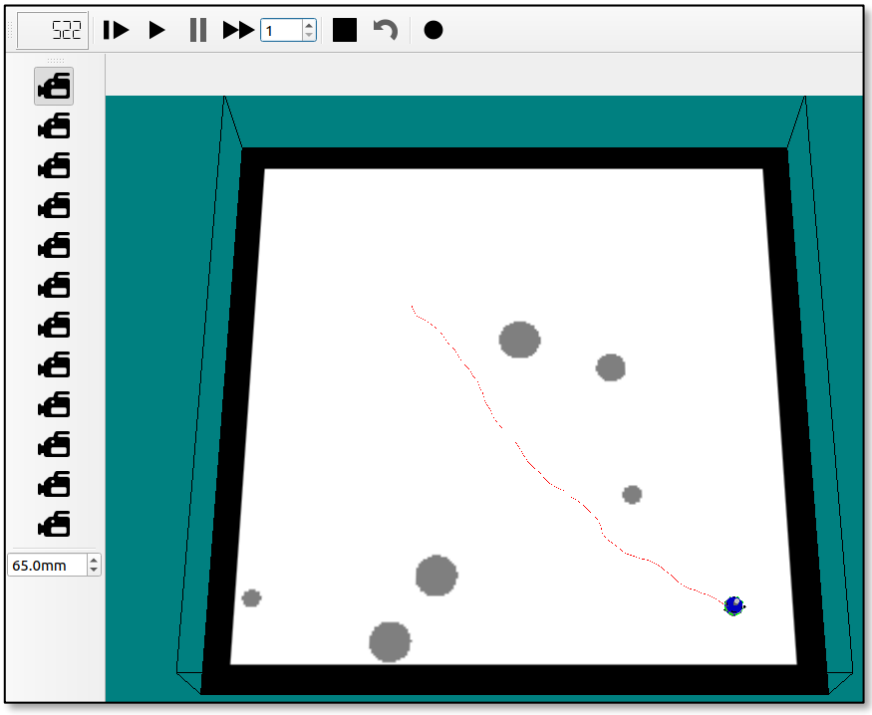
$Distance\ Metric = L1, k = 30$:



$Distance\ Metric = L2, k = 10$:



$Distance\ Metric = L2, k = 30$:

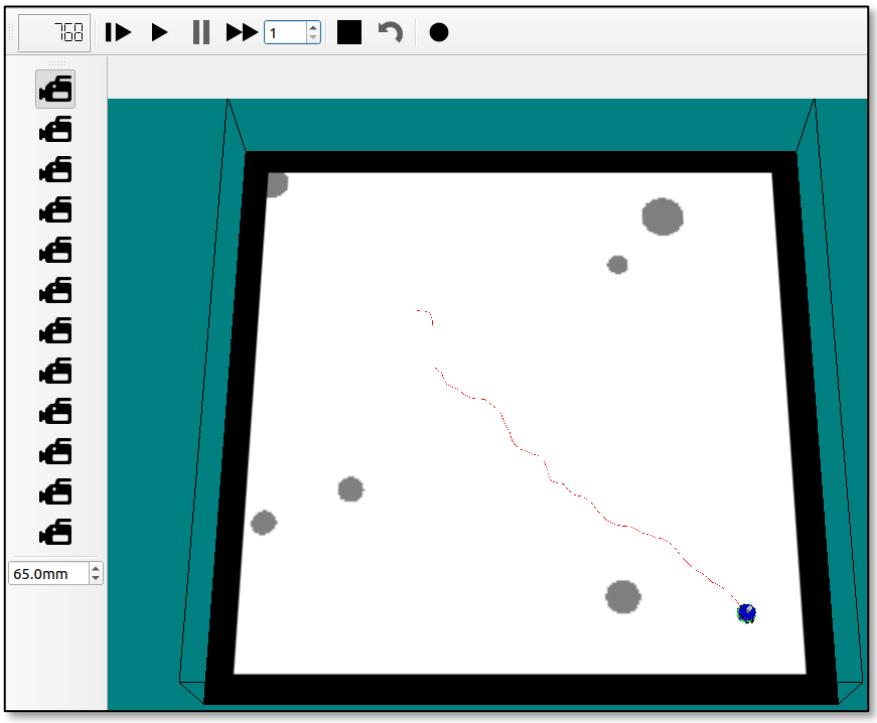


Insights: It seems that L2 metric yields better times, and it makes sense because the robot can drive in any angle, so the Euclidean Distance is more fit to this case. Moreover, it appears that the robot gets to the goal in slightly better time with $k = 30$, and that is because with $k = 10$ the neighbors than KNN finds are closer to the current position and as a result the robot advance in smaller steps (then in the case when $k = 30$).

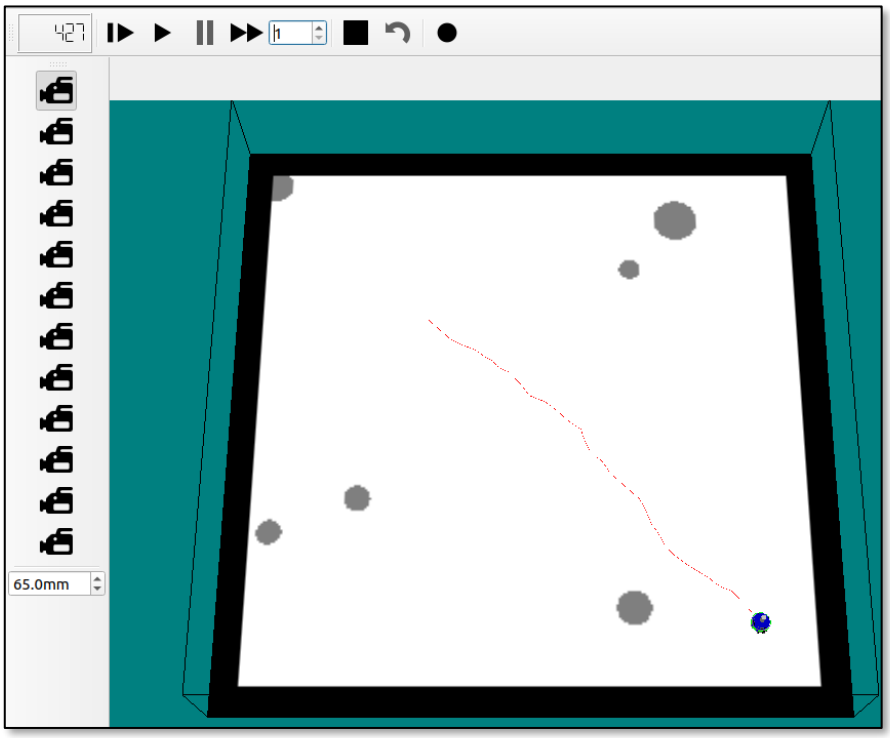
Seed Report 2:

In this experiment we set $random\ seed = \mathbf{3}$.

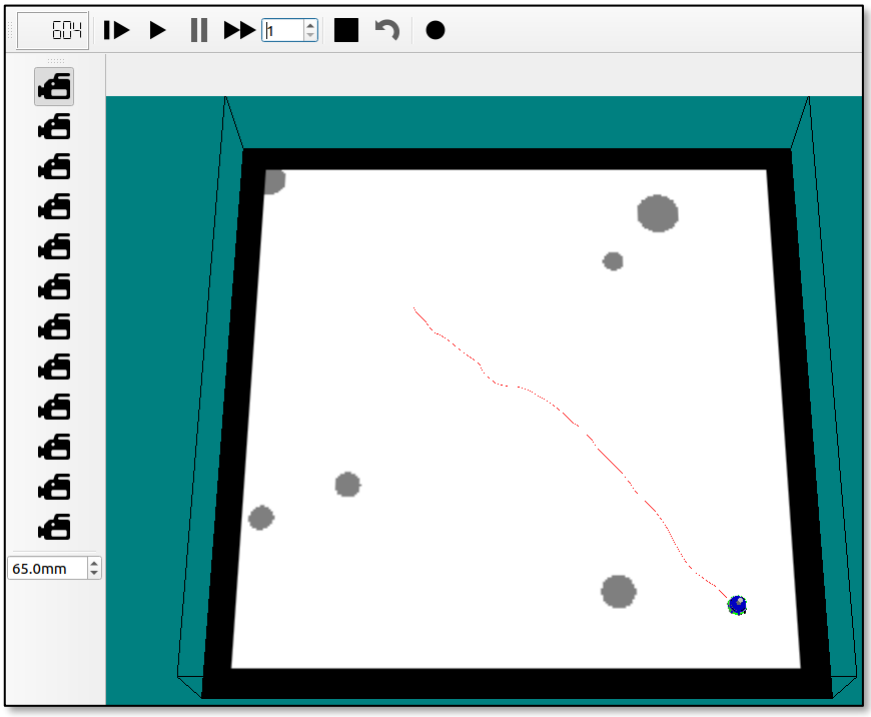
$Distance\ Metric = L1, k = 10$:



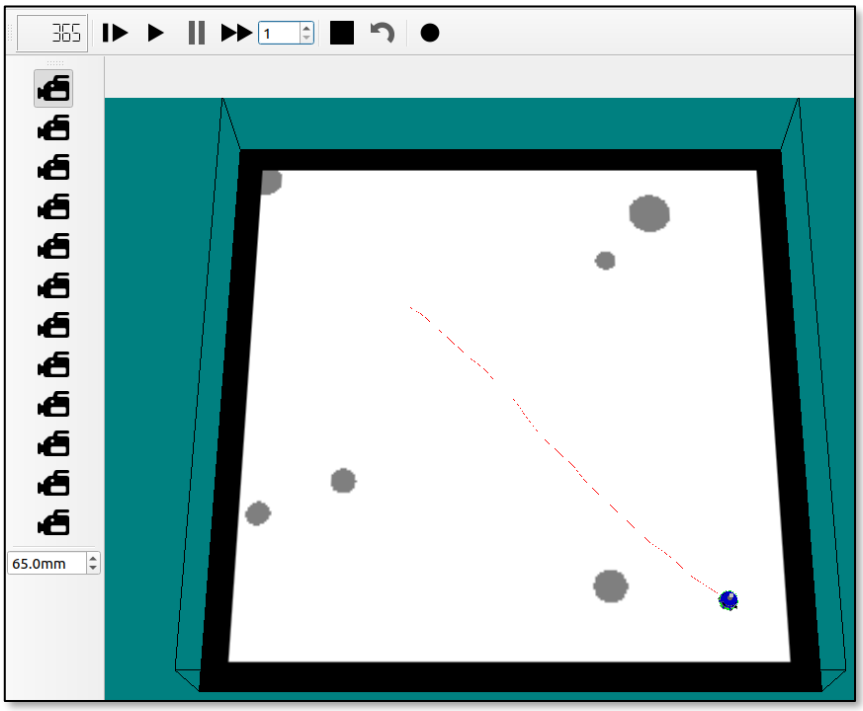
$Distance\ Metric = L1, k = 30$:



$Distance\ Metric = L2, k = 10$:



$Distance\ Metric = L2, k = 30$:

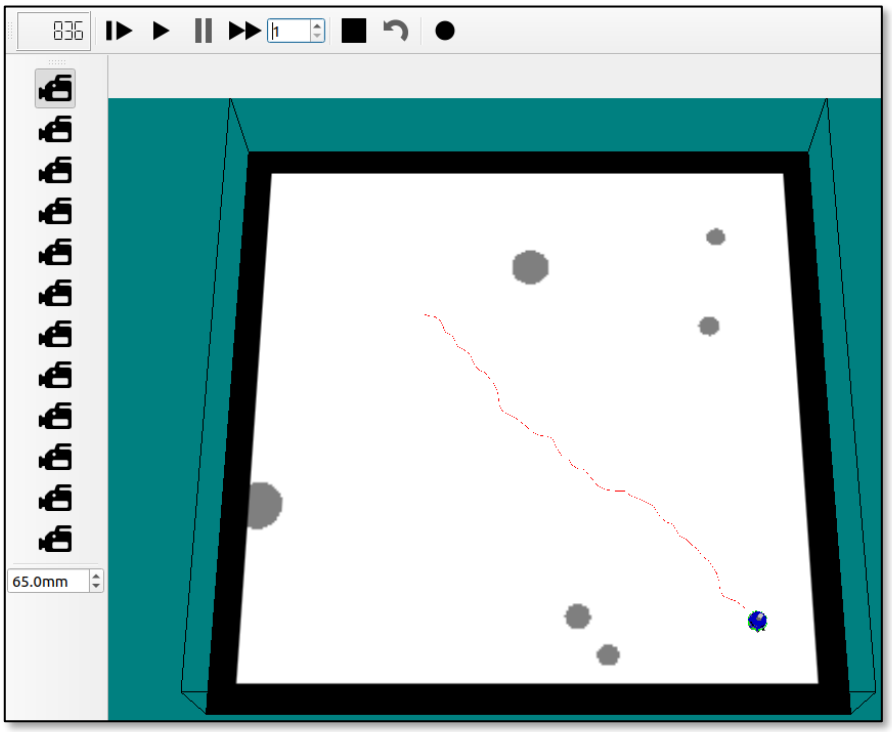


Insights: This experiment shows clearly how $k = 30$ is better then $k = 10$.

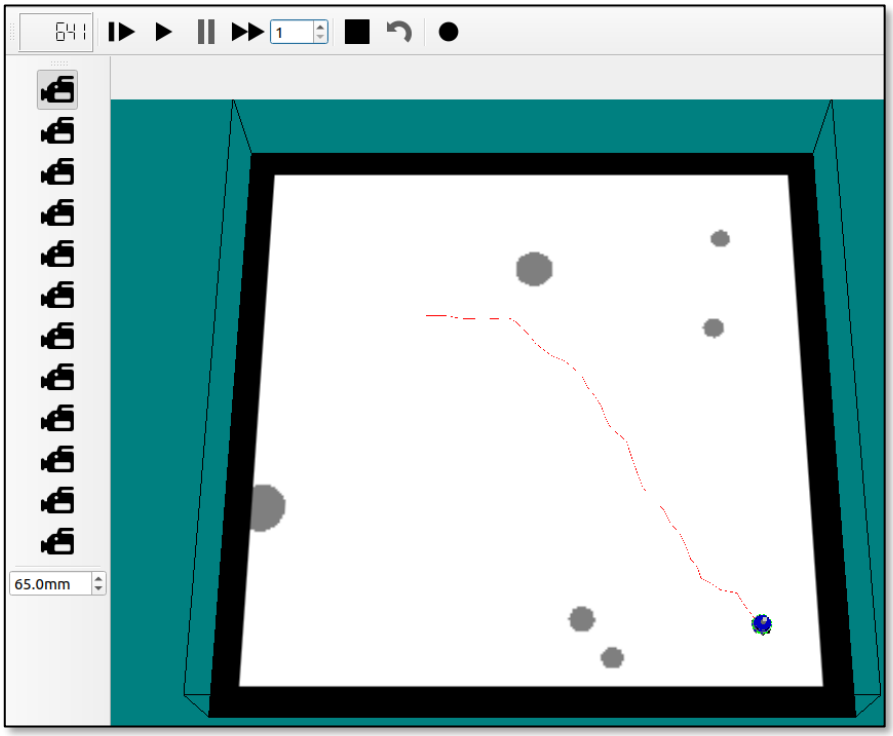
Seed Report 3:

In this experiment we set $random\ seed = \mathbf{6}$.

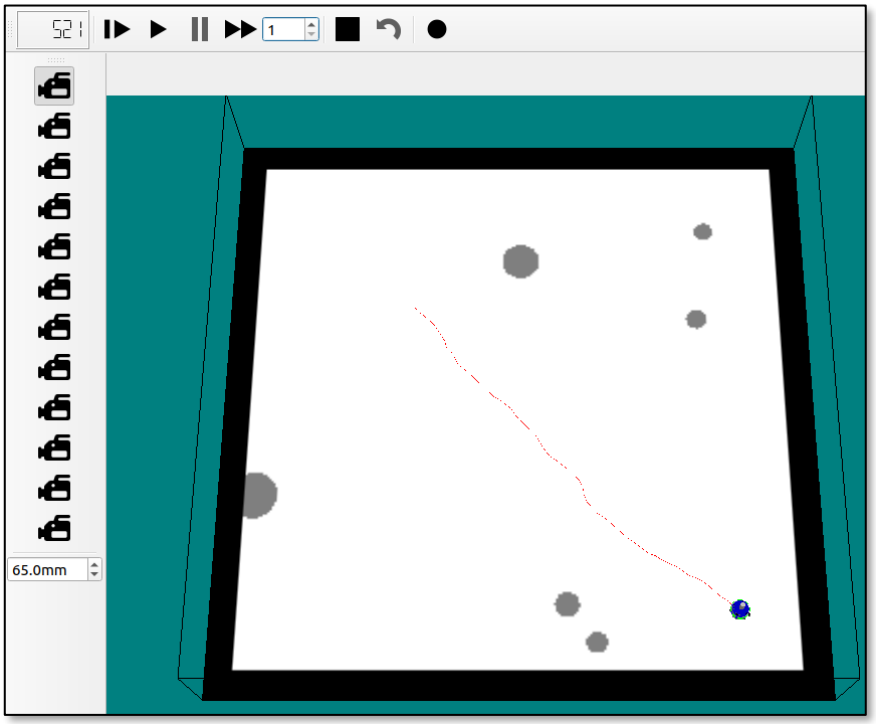
$Distance\ Metric = L1, k = 10$:



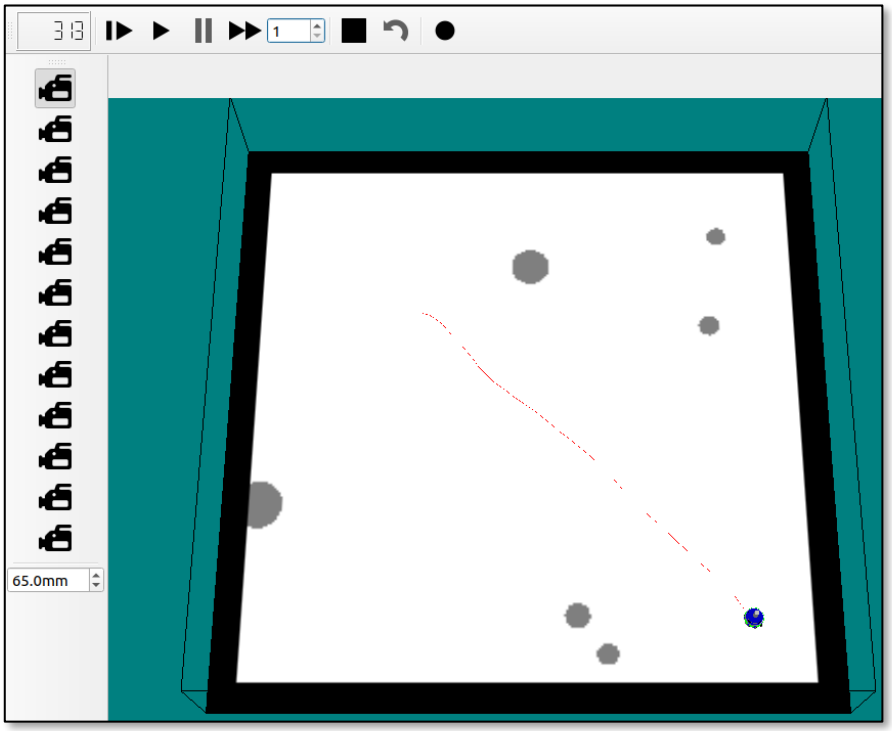
$Distance\ Metric = L1, k = 30$:



$Distance\ Metric = L2, k = 10$:



$Distance\ Metric = L2, k = 30$:

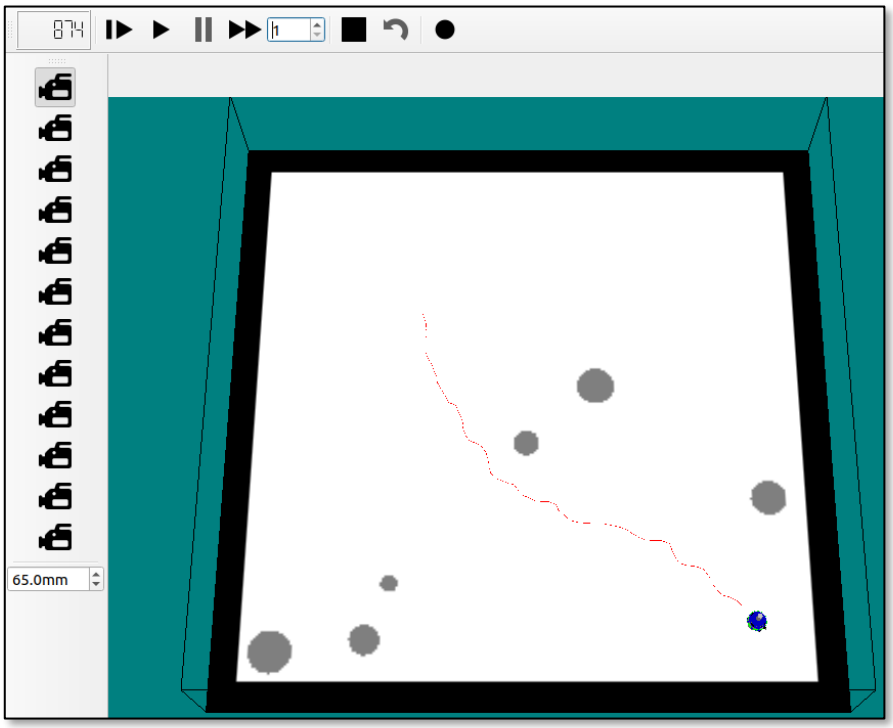


Insights: In this experiment there are no obstacles on the way from the robot's starting position to the goal, so we can expect the path's trajectory to be a straight as possible. We can see that the trajectory is straighter while using the L2 metric, so this experiment shows that is better to use L2 metric than L1.

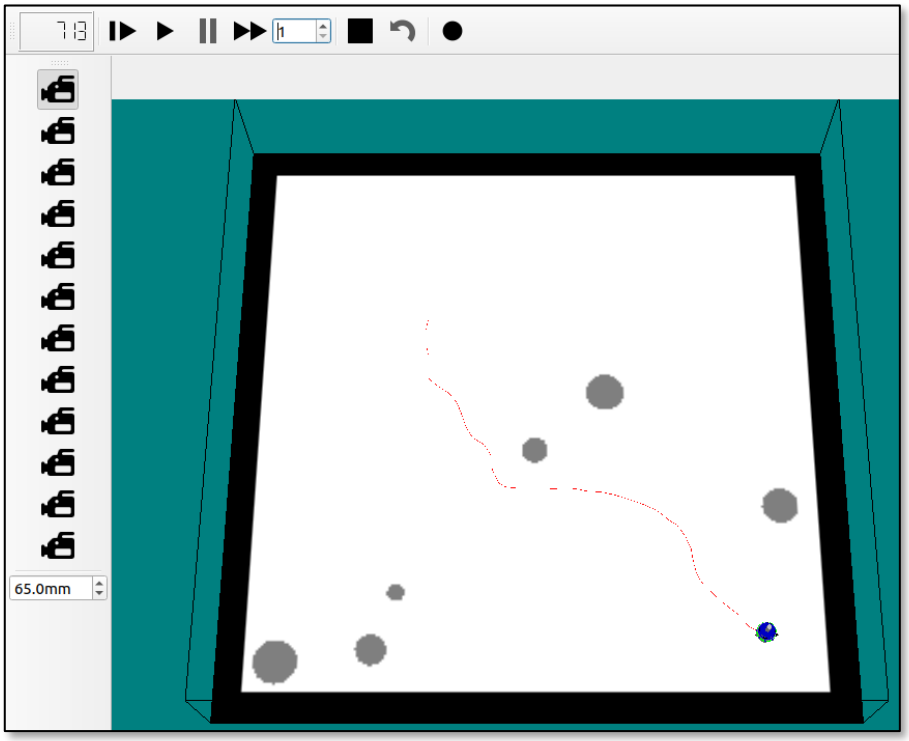
Seed Report 4:

In this experiment we set $random\ seed = \mathbf{8}$.

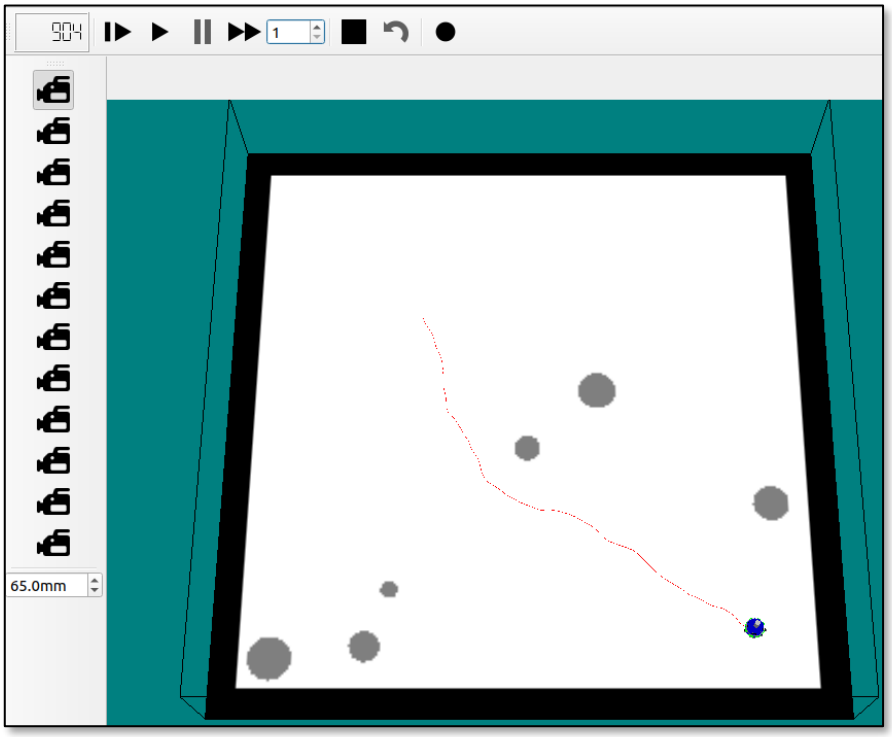
$Distance\ Metric = L1, k = 10$:



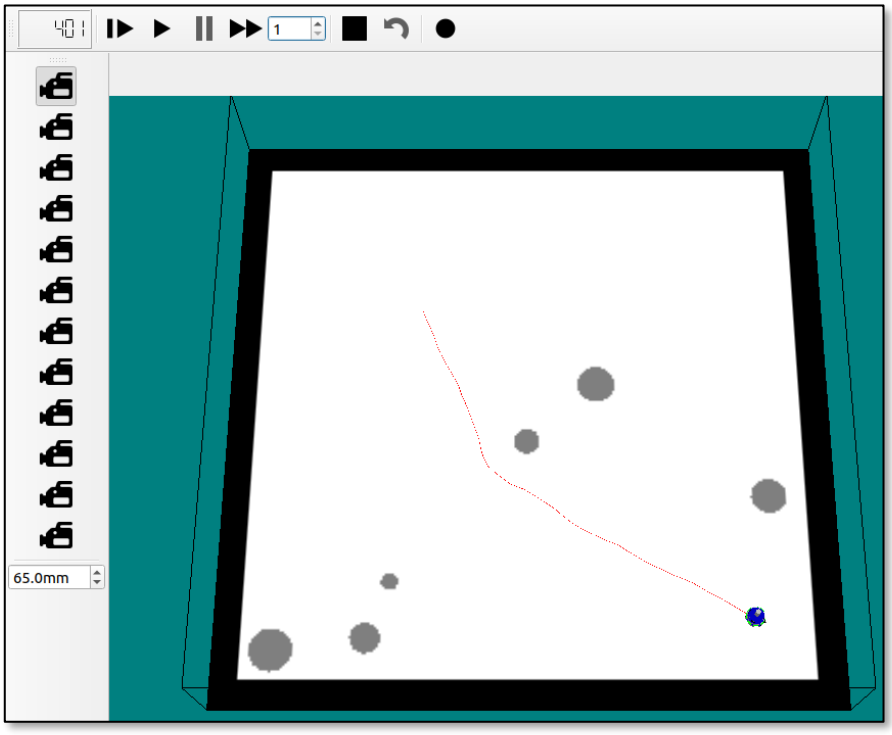
$Distance\ Metric = L1, k = 30$:



$Distance\ Metric = L2, k = 10$:



$Distance\ Metric = L2, k = 30$:

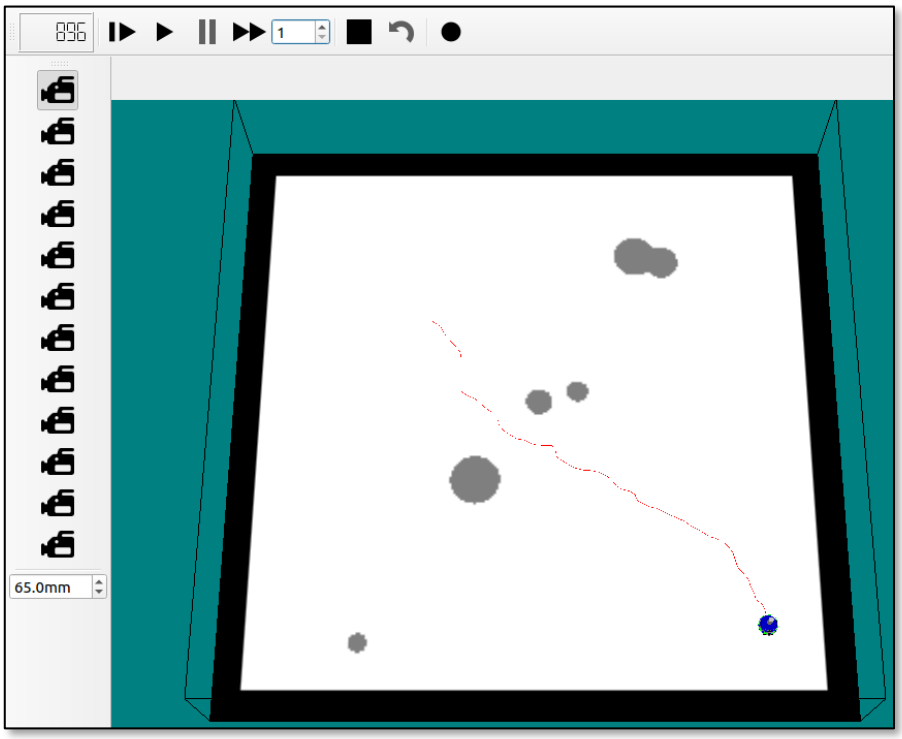


Insights: In this experiment there is only one obstacle on the way so we can claim that the optimal trajectory consists of two straight line – the first from the robot's starting position to the edge of the obstacle and the second from the edge to the goal. It appears that the trajectory most similar to the optimal trajectory is with L2 and $k = 30$.

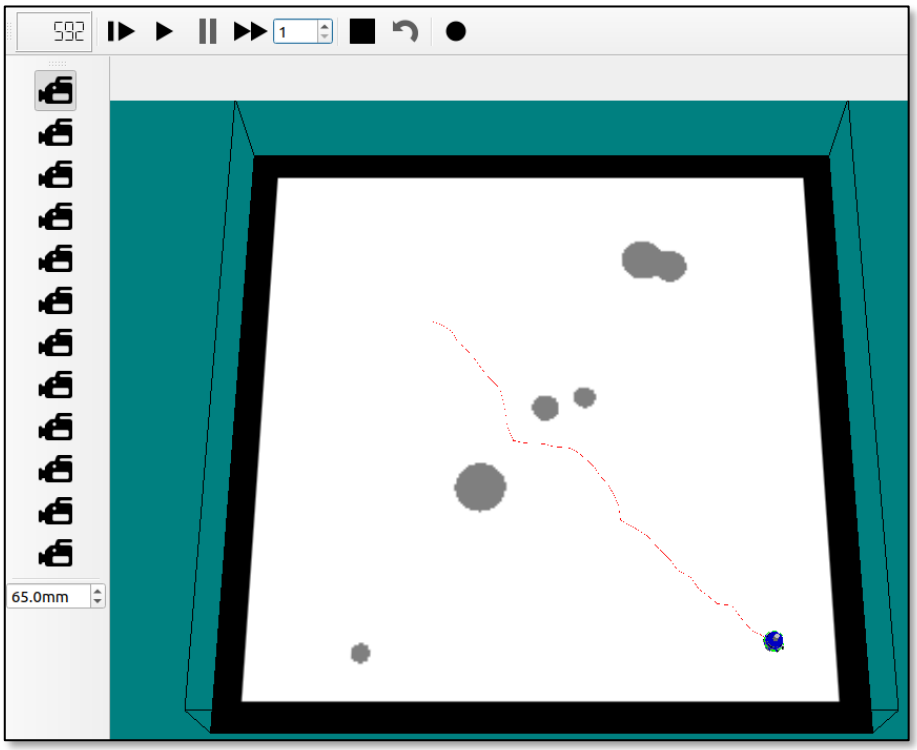
Seed Report 5:

In this experiment we set $random\ seed = \mathbf{16}$.

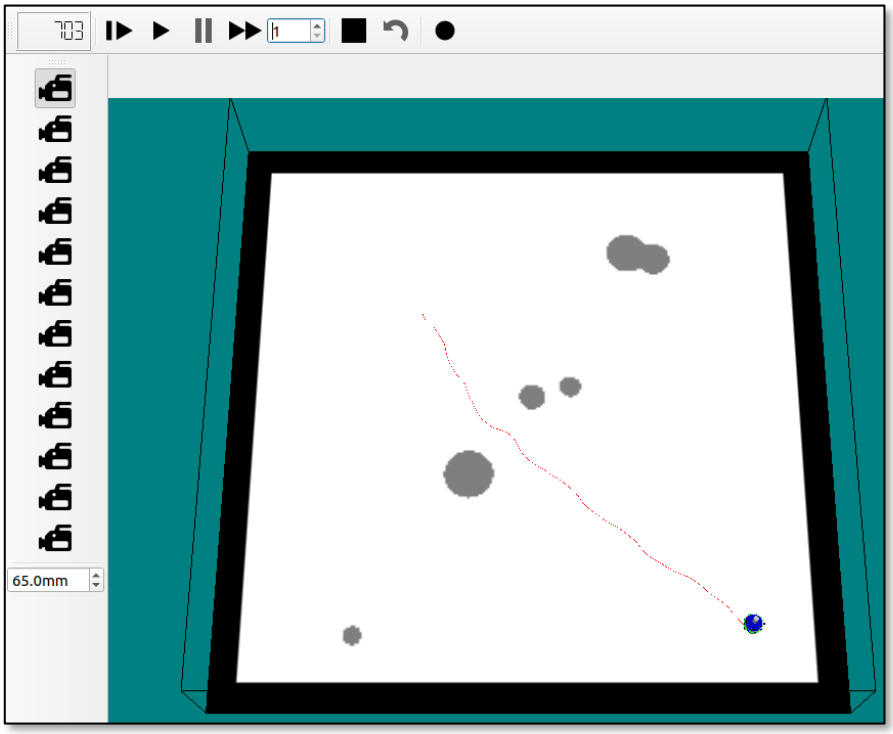
$Distance\ Metric = L1, k = 10$:



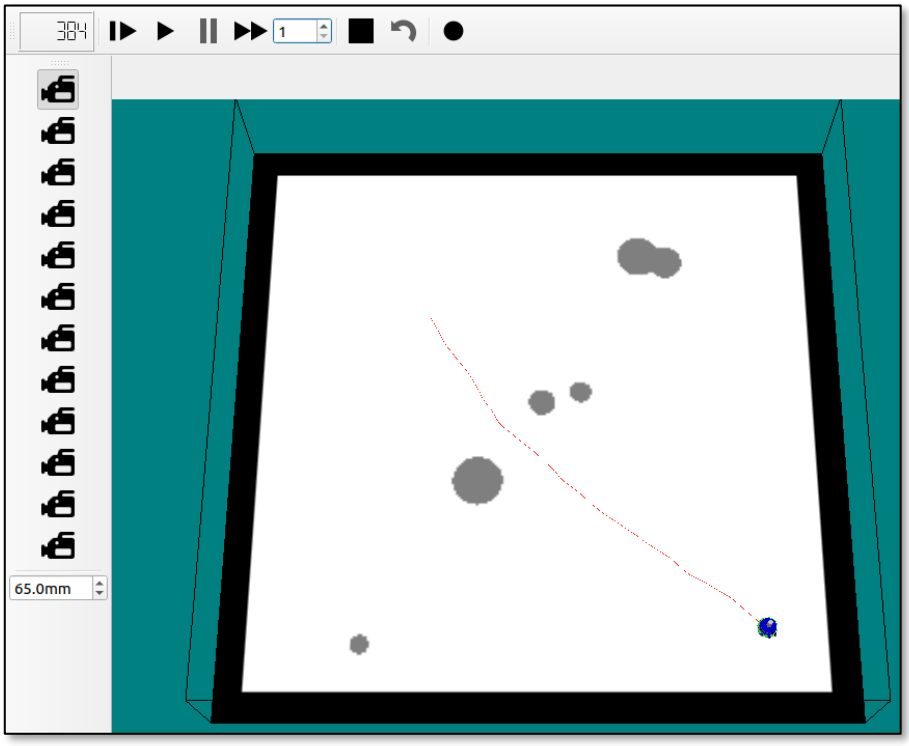
$Distance\ Metric = L1, k = 30$:



$Distance\ Metric = L2, k = 10$:



$Distance\ Metric = L2, k = 30$:



Insights: In this experiment we were looking for a seed that place multiple obstacles along the way from the robot's starting position to the goal, and we wanted to see how the robot is dogging the obstacles. It appears that the robot dogged the obstacles successfully with both metrics and both ks, but it seems that it does so in the most elegant way with L2 metric and $k = 30$.

Final Conclusion: our algorithm let the robot successfully navigate from the starting position to the goal in different arenas. We chose to use $KD\ Tree$ data structure for the KNN algorithm, and to find the shortest path between positions with $Dijkstra$ algorithm. Later, we have tested different values for each parameter in the algorithm and decided to finally determine them as follows:

- $Metric = L2\ Euclidean\ Distance$
- $k = 30$
- $Reduction\ Factor = 2$
- $Number\ of\ Samples = 12{,}000$
- $Error\ Angle = \pm 5°$
- $Angular\ Speed = 30$
- "$Touching\ Distance$" $= 2$ (The distance from waypoint to consider as "reached")